Unit 2

Random Search

The random search in machine learning involves generating and evaluating random inputs to the objective function. It is effective because it does not assume anything about the structure of the objective function. This can benefit problems with a lot of domain expertise that may influence or bias the optimization strategy, allowing non-intuitive solutions to be discovered. The drawback of random search is that it yields high variance during computing

The random search strategy consists of sampling solutions over the entire search space using a uniform probability distribution. Each future sample is independent of the samples that precede it.

The strategy has a complexity time and minimal memory, as it requires only a candidate solution construction routine and a candidate solution evaluation routine, both of which can be calibrated using the approach.

The worst performance for locating optima is worse than a search domain enumeration, since the random search has no memory and can perform blind resampling.

The Algorithm

So, how does random search work? The search begins by initializing random hyper parameter values from the search space. Let's call this point x. Next, it calculates the value of the cost function at x. Then it takes another set of random hyper parameter values, let's call it y, and calculates the value of the cost function at y. If the value of the cost function at y is lower than that at x, then it repeats these steps by assigning x = y as the new start. Else it continues from x. This process is repeated until a requirement set by the user still needs to be fulfilled.

The following steps describe the algorithm of random search in machine learning.

1. Set x to a random place in the search space.
2. Repeat until a termination requirement, such as multiple iterations completed or appropriate fitness achieved, is met:
    - Take a new position y from the hyper sphere with a given radius around the current position x.
    - If f(y) < f(x), then assign x = y as the new position.

Advantages of Random Search

These are the few advantages of random search in machine learning over grid search.

- Random search gives better results than grid search when the dimensionality and the number of hyper parameters are high.
- Random search in machine learning allows us to limit the number of hyper parameter combinations. Whereas in grid search, all the varieties of hyper parameters are checked.
- The random search usually gives better results than the grid search in less iteration.

Closed and open list

Closed list describes the variant of party-list systems where voters can effectively vote for only political parties as a whole; thus they have no influence on the party-supplied order in which party candidates are elected. If voters had some influence, that would be called an open list.

Uninformed Search Techniques

Depth First Search

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.

- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

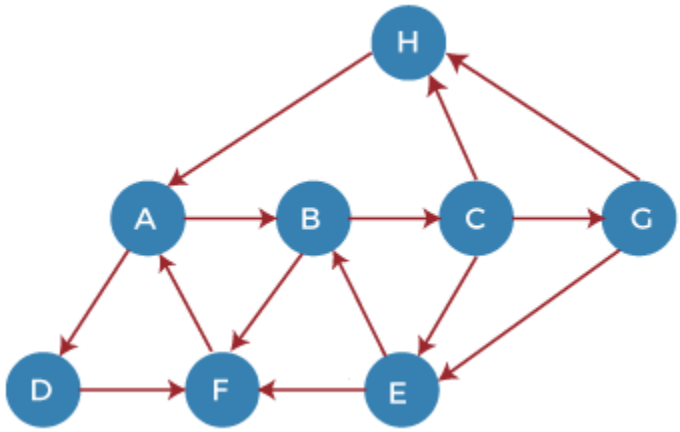
Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

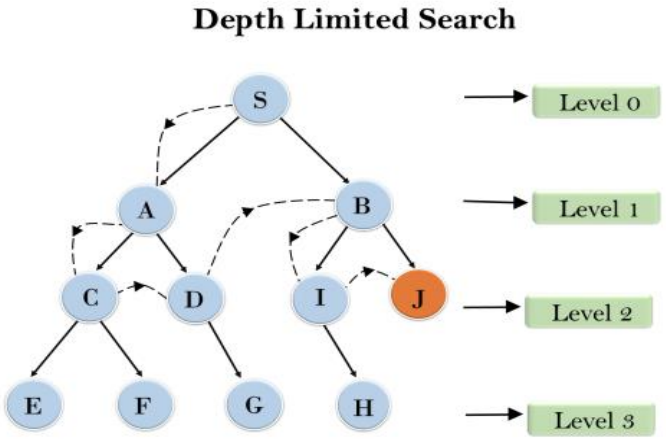
The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})** where b is the branching factor of the search tree, and l is the depth limit.

Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)** where **b** is the branching factor of the search tree, and **l** is the depth limit.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

Breadth First Search algorithm

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes.
- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.
- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

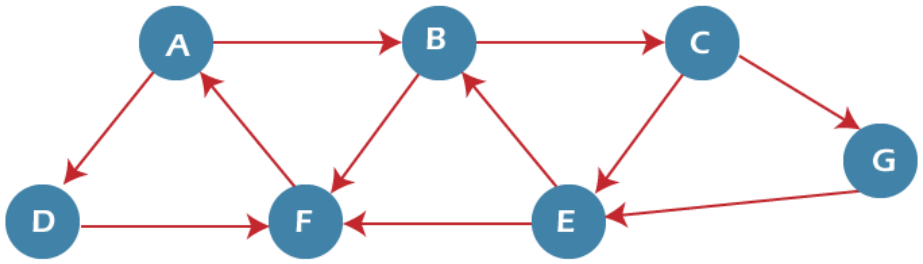
(waiting state)

[END OF LOOP]

Step 6: EXIT

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Complexity of BFS algorithm

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is **O(V+E)**, since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

Uniform Cost Search (UCS)

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Key Concepts of Uniform Cost Search

1. **Priority Queue:** UCS uses a priority queue to store nodes. The node with the lowest cumulative cost is expanded first. This ensures that the search explores the most promising paths first.

2. **Path Cost:** The cost associated with reaching a particular node from the start node. UCS calculates the cumulative cost from the start node to the current node and prioritizes nodes with lower costs.

3. **Exploration:** UCS explores nodes by expanding the least costly node first, continuing this process until the goal node is reached. The path to the goal node is guaranteed to be the least costly one.

4. **Termination:** The algorithm terminates when the goal node is expanded, ensuring that the first time the goal node is reached, the path is the optimal one**.**

Working of Uniform Cost Search

UCS operates under a simple principle: among all possible expansions, pick the path that has the smallest total cost from the start node. This is implemented using a priority queue to keep the partial paths in order, based on the total cost from the root node.

Following steps are performed to execute UCS:

1. **Initialization:** UCS starts with the root node. It is added to the priority queue with a cumulative cost of zero since no steps have been taken yet.

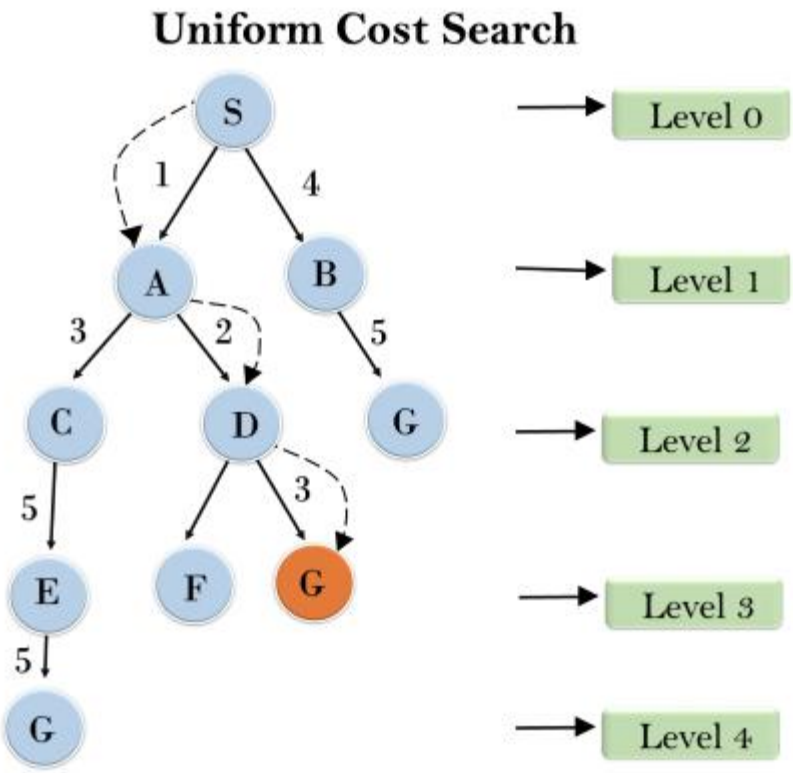
2. **Node Expansion:** The node with the lowest path cost is removed from the priority queue. This node is then expanded, and its neighbors are explored.

3. **Exploring Neighbors:** For each neighbor of the expanded node, the algorithm calculates the total cost from the start node to the neighbor through the current node. If a neighbor node is not in the priority queue, it is added to the queue with the calculated cost. If the neighbor is already in the queue but a lower cost path to this neighbor is found, the cost is updated in the queue.

4. **Goal Check:** After expanding a node, the algorithm checks if it has reached the goal node. If the goal is reached, the algorithm returns the total cost to reach this node and the path taken.

5. **Repetition:** This process repeats until the priority queue is empty or the goal is reached.

   Example:



Iterative deepeningdepth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Here are the steps for Iterative deepening depth first search algorithm:

- Set the depth limit to 0.
- Perform DFS to the depth limit.
- If the goal state is found, return it.
- If the goal state is not found and the maximum depth has not been reached, increment the depth limit and repeat steps 2-4.
- If the goal state is not found and the maximum depth has been reached, terminate the search and return failure.

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- It is a type of straightforward which is used to put into practice since it builds upon the conventional depth-first search algorithm.
- It is a type ofsearch algorithm which provides guarantees to find the optimal solution, as long as the cost of each edge in the search space is the same.
- It is a type ofcomplete algorithm, and the meaning of this is it will always find a solution if one exists.
- The Iterative Deepening Depth-First Search (IDDFS) algorithm uses less memory compared to Breadth-First Search (BFS) because it only stores the current path in memory, rather than the entire search tree.

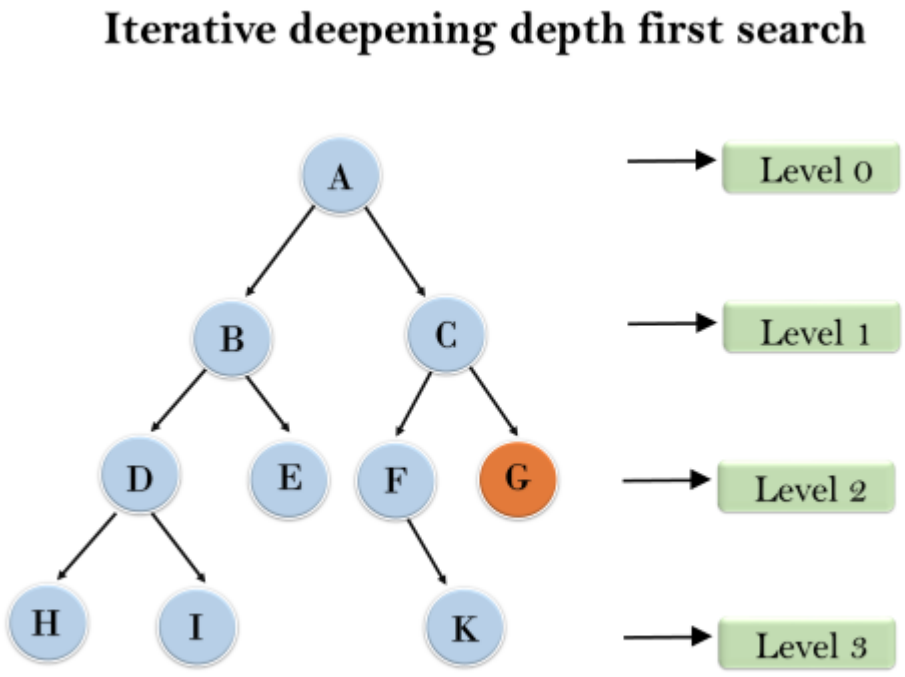
Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Example:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory
- The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.
- The cost of expanding nodes can be high in certain cases. In such scenarios, using this approach can help reduce the number of nodes that need to be expanded.

Disadvantages:

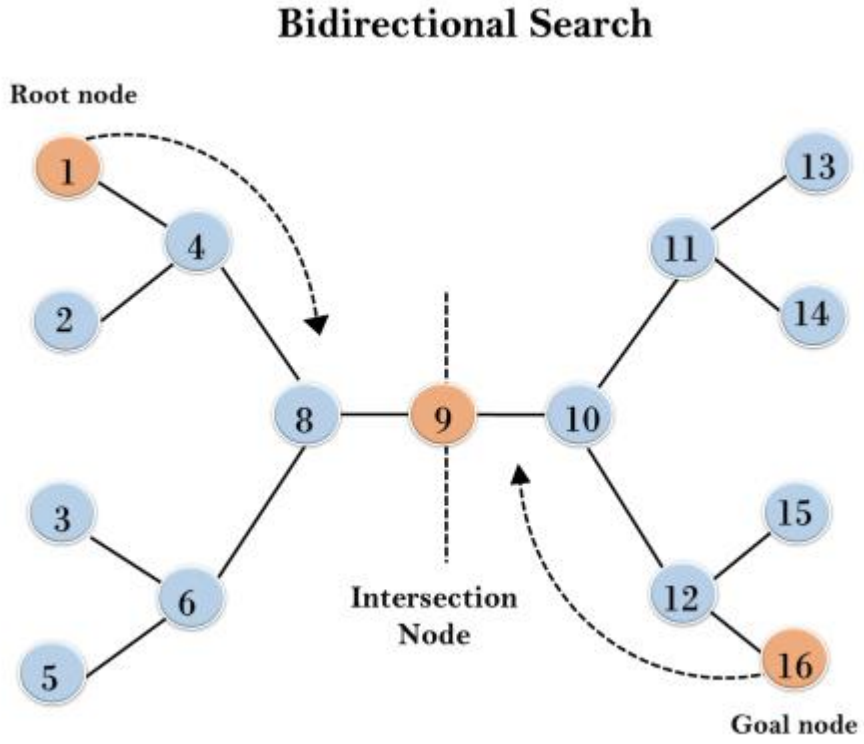
- Implementation of the bidirectional search tree is difficult.

- In bidirectional search, one should know the goal state in advance.
- Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

Space Complexity: Space complexity of bidirectional search is **$O(b^d)$**.

Optimal: Bidirectional search is Optimal.

Heuristic Search

Heuristics operates on the search space of a problem to find the best or closest-to-optimal solution via the use of systematic algorithms. A heuristic search method uses heuristic information to define a route that seems more plausible than the rest. Heuristics, in this case, refer to a set of criteria or rules of thumb that offer an estimate of a firm's profitability. Utilizing heuristic guiding, the algorithms determine the balance between exploration and exploitation, and thus they can successfully tackle demanding issues. Therefore, they enable an efficient solution finding process.

A heuristic is a technique that is used to solve a problem faster than the classic methods. These techniques are used to find the approximate solution of a problem when classical methods do not. Heuristics are said to be the problem-solving techniques that result in practical and quick solutions.

Components of Heuristic Search

Heuristic search algorithms typically comprise several essential components:

1. **State Space:** This implies that the totality of all possible states or settings, which is considered to be the solution for the given problem.
2. **Initial State:** The instance in the search tree of the highest level with no null values, serving as the initial state of the problem at hand.
3. **Goal Test:** The exploration phase ensures whether the present state is a terminal or consenting state in which the problem is solved.
4. **Successor Function:** This create a situation where individual states supplant the current state which represent the possible moves or solutions in the problem space.
5. **Heuristic Function:** The function of a heuristic is to estimate the value or distance from a given state to the target state. It helps to focus the process on regions or states that has prospect of achieving the goal.

Applications of Heuristic Search

Heuristic search techniques find application in a wide range of problem-solving scenarios, including:

1. **Pathfinding:** Discovery, of the shortest distance that can be found from the start point to the destination at the point of coordinates or graph.
2. **Optimization:** Solving the problem of the optimal distribution of resources, planning or posting to achieve maximum results.
3. **Game Playing:** The agency of AI with some board games, e.g., chess or Go, is on giving guidance and making strategy-based decisions to the agents.
4. **Robotics:** Scheduling robots` location and movement to guide carefully expeditions and perform given tasks with high efficiency.
5. **Natural Language Processing:** Language processing tasks involving search algorithms, such as parsing or semantic analysis, should be outlined. That means.

Advantages of Heuristic Search Techniques

Heuristic search techniques offer several advantages:

1. **Efficiency:** As they are capable of aggressively digesting large areas for the more promising lines, they can allot more time and resources to investigate the area.
2. **Optimality:** If the methods that an algorithm uses are admissible, A* guarantees of an optimal result.
3. **Versatility:** Heuristic search methods encompass a spectrum of problems that are applied to various domains of problems.

Limitations of Heuristic Search Techniques

1. **Heuristic Quality:** The power of heuristic search strongly depends on the quality of function the heuristic horizon. If the heuristics are constructed thoughtlessly, then their level of performance may be low or inefficient.
2. **Space Complexity:** The main requirement for some heuristic search algorithms could be a huge memory size in comparison with the others, especially in cases where the search space considerably increases.
3. **Domain-Specificity:** It is often the case that devising efficient heuristics depends on the specifics of the domain, a challenging obstruction to development of generic approaches.

Informed Search Algorithms

Greedy-Best-first search algorithm

Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.

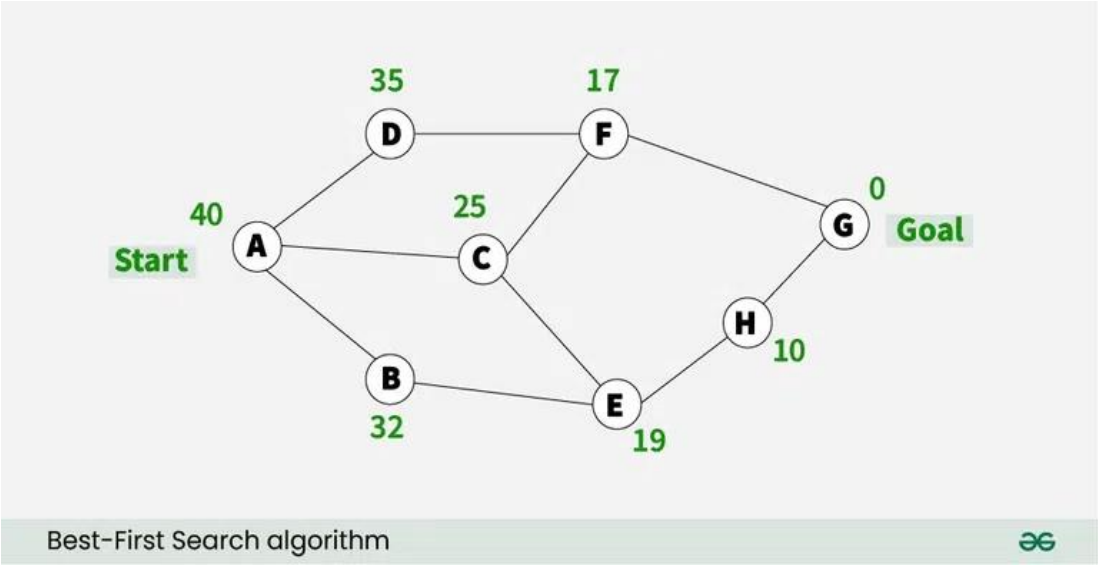
The algorithm works by using a heuristic function to determine which path is the most promising. The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths. If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

Working of Greedy Best-First Search

- Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a heuristic function to determine which path is the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

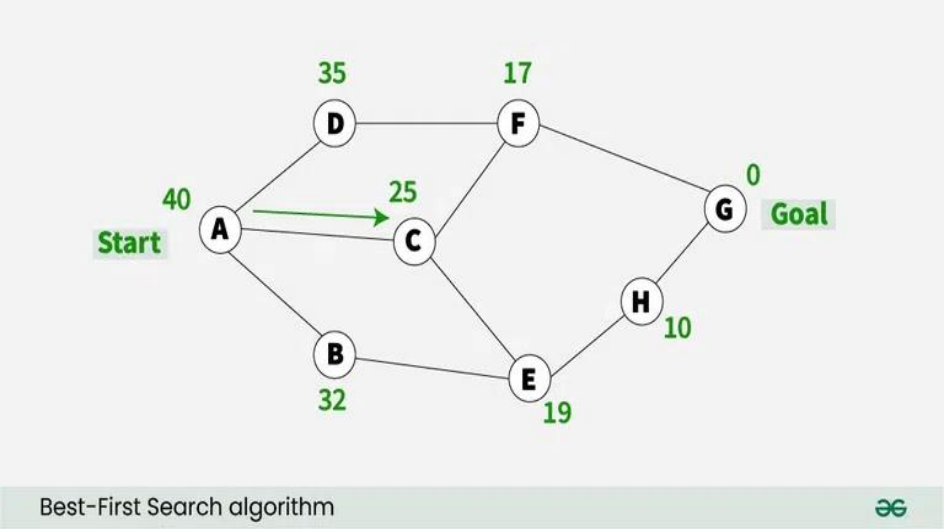
  Example :

An example of the best-first search algorithm is below graph, suppose we have to find the path from A to G

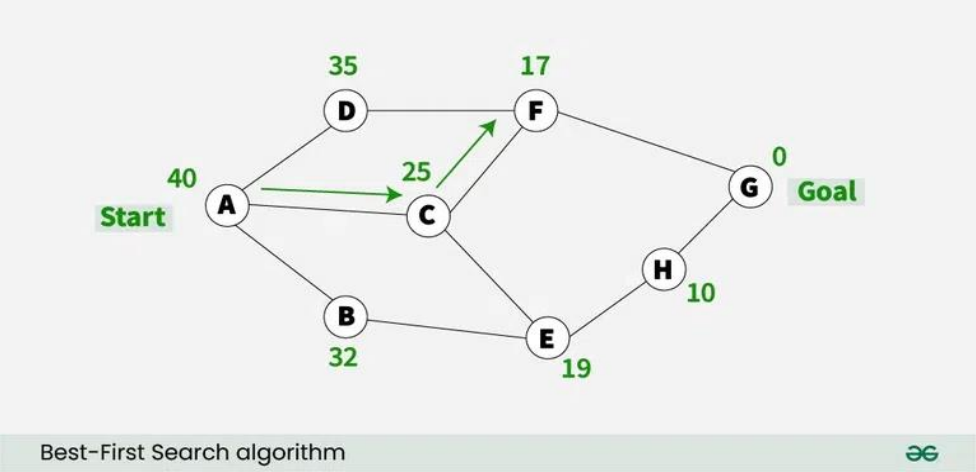


1) We are starting from A , so from A there are direct path to node B(with heuristics value of 32) , from A to C (with heuristics value of 25) and from A to D(with heuristics value of 35) .

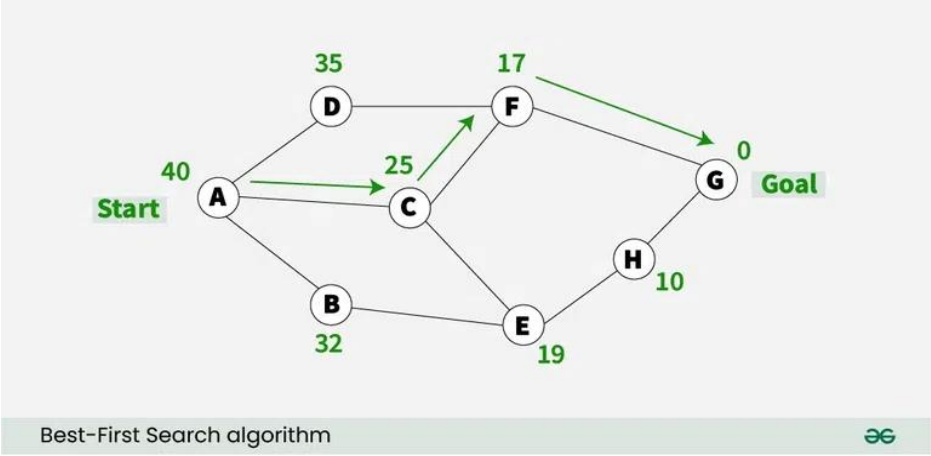
2) So as per best first search algorithm choose the path with lowest heuristics value , currently C has lowest value among above node . So we will go from A to C.



3) Now from C we have direct paths as C to F(with heuristics value of 17) and C to E(with heuristics value of 19) , so we will go from C to F.



4) Now from F we have direct path to go to the goal node G (with heuristics value of 0) , so we will go from F to G.



*5) So now the goal node G has been reached and the path we will follow is **A->C->F->G** .*

Advantages of Greedy Best-First Search:

- **Simple and Easy to Implement:** Greedy Best-First Search is a relatively straightforward algorithm, making it easy to implement.
- **Fast and Efficient:** Greedy Best-First Search is a very fast algorithm, making it ideal for applications where speed is essential.
- **Low Memory Requirements:** Greedy Best-First Search requires only a small amount of memory, making it suitable for applications with limited memory.
- **Flexible:** Greedy Best-First Search can be adapted to different types of problems and can be easily extended to more complex problems.
- **Efficiency:** If the heuristic function used in Greedy Best-First Search is good to estimate, how close a node is to the solution, this algorithm can be a very efficient and find a solution quickly, even in large search spaces.

Disadvantages of Greedy Best-First Search:

- **Inaccurate Results:** Greedy Best-First Search is not always guaranteed to find the optimal solution, as it is only concerned with finding the most promising path.
- **Local Optima:** Greedy Best-First Search can get stuck in local optima, meaning that the path chosen may not be the best possible path.
- **Heuristic Function:** Greedy Best-First Search requires a heuristic function in order to work, which adds complexity to the algorithm**.**
- **Lack of Completeness:** Greedy Best-First Search is not a complete algorithm, meaning it may not always find a solution if one is exists. This can happen if the algorithm gets stuck in a cycle or if the search space is a too much complex.

Applications of Greedy Best-First Search:

- **Pathfinding:** Greedy Best-First Search is used to find the shortest path between two points in a graph. It is used in many applications such as video games, robotics, and navigation systems.
- **Machine Learning:** Greedy Best-First Search can be used in machine learning algorithms to find the most promising path through a search space.
- **Optimization:** Greedy Best-First Search can be used to optimize the parameters of a system in order to achieve the desired result.

- **Game AI:** Greedy Best-First Search can be used in game AI to evaluate potential moves and chose the best one.
- **Navigation:** Greedy Best-First Search can be use to navigate to find the shortest path between two locations.
- **Natural Language Processing:** Greedy Best-First Search can be use in natural language processing tasks such as language translation or speech recognisation to generate the most likely sequence of words.
- **Image Processing:** Greedy Best-First Search can be use in image processing to segment image into regions of interest.

A* Search Algorithm

A* (pronounced "A-star") is a powerful graph traversal and pathfinding algorithm widely used in artificial intelligence and computer science. It is mainly used to find the shortest path between two nodes in a graph, given the estimated cost of getting from the current node to the destination node. The main advantage of the algorithm is its ability to provide an optimal path by exploring the graph in a more informed way compared to traditional search algorithms such as Dijkstra's algorithm.

Algorithm A* combines the advantages of two other search algorithms: Dijkstra's algorithm and Greedy Best-First Search. Like Dijkstra's algorithm, A* ensures that the path found is as short as possible but does so more efficiently by directing its search through a heuristic similar to Greedy Best-First Search. A heuristic function, denoted h(n), estimates the cost of getting from any given node n to the destination node

The main idea of A* is to evaluate each node based on two parameters:

1. **g(n):** the actual cost to get from the initial node to node n. It represents the sum of the costs of node n outgoing edges.
2. **h(n):** Heuristic cost (also known as "estimation cost") from node n to destination node n. This problem-specific heuristic function must be acceptable, meaning it never overestimates the actual cost of achieving the goal. The evaluation function of node n is defined as f(n) = g(n) h(n).

Working of AO* algorithm:

The evaluation function in AO* looks like this:

f(n) = g(n) + h(n)

f(n) = Actual cost + Estimated cost

here,

f(n) = The actual cost of traversal.

g(n) = the cost from the initial node to the current node.
h(n) = estimated cost from the current node to the goal state.

Algorithm A* selects the nodes to be explored based on the lowest value of f(n), preferring the nodes with the lowest estimated total cost to reach the goal. The A* algorithm works:

1. Create an open list of foundbut not explored nodes.
2. Create a closed list to hold already explored nodes.
3. Add a startingnode to the open list with an initial value of g
4. Repeat the following steps until the open list is empty or you reachthe target node:

a. Find the node with the smallest f-value (i.e., the node with the minor g(n) h(n)) in the open list.

b. Move the selected node from the open list to the closed list.

c. Createall valid descendantsof the selected node.

d. For each successor, calculateits g-value as the sum of the current node's g value and the cost of movingfrom the current node to the successor node. Update the g-value of the tracker when a better path is found.

e. If the followeris not in the open list, add it with the calculated g-value and calculate its h-value. If it is already in the open list, update its g value if the new path is better.

f. Repeat the cycle. Algorithm A* terminates when the target node is reached or when the open list empties, indicating no paths from the start node to the target node. The A* search algorithm is widely used in various fields such as robotics, video games, network routing, and design problems because it is efficient and can find optimal paths in graphs or networks.

Advantages of A* Search Algorithm

1. **Optimal solution:** A* ensures finding the optimal (shortest) path from the start node to the destination node in the weighted graph given an acceptable heuristic function. This optimality is a decisive advantage in many applications where finding the shortest path is essential.

2. **Completeness:** If a solution exists, A* will find it, provided the graph does not have an infinite cost This completeness property ensures that A* can take advantage of a solution if it exists.

3. **Efficiency:** A* is efficient ifan efficient and acceptable heuristic function is used. Heuristics guide the search to a goal by focusing on promising paths and avoiding unnecessary exploration, making A* more efficient than non-aware search algorithms such as breadth-first search or depth-first search.

4. **Versatility:** A* is widely applicable to variousproblem areas, including wayfinding, route planning, robotics, game development, and more. A* can be used to find optimal solutions efficiently as long as a meaningful heuristic can be defined.

5. **Optimized search:** A* maintains a priority order to select the nodes with the minor f(n) value (g(n) and h(n)) for expansion. This allows it to explore promising paths first, which reduces the search space and leads to faster convergence.

6. **Memory efficiency:** Unlike some other search algorithms, such as breadth-first search, A* stores only a limited number of nodes in the priority queue, which makes it memory efficient, especially for large graphs.

7. **Tunable Heuristics:** A*'s performancecan be fine-tuned by selecting different heuristic functions. More educated heuristics can lead to faster convergence and less expanded nodes.

8. **Extensively researched:** A* is a well-established algorithm with decades of research and practical applications. Many optimizations and variations have been developed, making it a reliable and well-understood troubleshooting tool.

9. **Web search:** A* can be used for web-based path search, where the algorithm constantly updates the path according to changes in the environment or the appearance of new It enables real-time decision-making in dynamic scenarios.

Disadvantages of A* Search Algorithm

1. **Heuristic accuracy:** The performance of the A* algorithm depends heavily on the accuracy of the heuristic function used to estimate the cost from the current node to the If the heuristic is unacceptable (never overestimates the actual cost) or inconsistent (satisfies the triangle inequality), A* may not find an optimal path or may explore more nodes than necessary, affecting its efficiency and accuracy.

2. **Memory usage:** A* requires that all visited nodes be kept in memory to keep track of explored paths. Memory usage can sometimes become a significant

issue, especially when dealing with an ample search space or limited memory resources.

3. **Time complexity:** AlthoughA* is generally efficient, its time complexity can be a concern for vast search spaces or graphs. In the worst case, A* can take exponentially longer to find the optimal path if the heuristic is inappropriate for the problem.
4. **Bottleneck at the destination:** In specific scenarios, the A* algorithm needs to explore nodes far from the destination before finally reaching the destination region. This the problem occurs when the heuristic needs to direct the search to the goal early effectively.
5. **Cost Binding:** A* faces difficulties when multiple nodes have the same f-value (the sum of the actual cost and the heuristic cost). The strategy used can affect the optimality and efficiency of the discovered path. If not handled correctly, it can lead to unnecessary nodes being explored and slow down the algorithm.
6. **Complexity in dynamic environments:** In dynamic environments where the cost of edges or nodes may change during the search, A* may not be suitable because it does not adapt well to such changes. Reformulation from scratch can be computationally expensive, and D* (Dynamic A*) algorithms were designed to solve this
7. **Perfection in infinite space :** A* may not find a solution in infinite state space. In such cases, it can run indefinitely, exploring an ever-increasing number of nodes without finding a solution. Despite these shortcomings, A* is still a robust and widely used algorithm because it can effectively find optimal paths in many practical situations if the heuristic function is well-designed and the search space is manageable. Various variations and variants of A* have been proposed to alleviate some of its limitations.
8. **Pathfinding in Games:** A* is oftenused in video games for character movement, enemy Al navigation, and finding the shortest path from one location to another on the game map. Its ability to find the optimal path based on cost and heuristics makes it ideal for real-time applications such as games.
9. **Robotics and Autonomous Vehicles:** A* is used in robotics and autonomous vehicle navigation to plan anoptimal route for robots to reach a destination, avoiding obstacles and considering terrain costs. This is crucial for efficient and safe movement in natural environments.

10. **Maze solving:** A* can efficiently find the shortest path through a maze, making it valuable in many maze-solving applications, such as solving puzzles or navigating complex structures.
11. **Route planningand navigation:** In GPS systems and mapping applications, A* can be used to find the optimal route between two points on a map, considering factors such as distance, traffic conditions, and road network topology.
12. **Puzzle-solving:** A* can solve various diagram puzzles, such as sliding puzzles, Sudoku, and the 8-puzzle problem. Resource Allocation: In scenarios where resources must be optimally allocated, A* can help find the most efficient allocation path, minimizing cost and maximizing efficiency.
13. **Network Routing:** A* can be usedin computer networks to find the most efficient route for data packets from a source to a destination node.
14. **Natural Language Processing (NLP):** In some NLP tasks, A* can generate coherent and contextualresponses by searching for possible word sequences based on their likelihood and relevance.
15. **Path planningin robotics:** A* can be used to plan the path of a robot from one point to another, considering various constraints, such as avoiding obstacles or minimizing energy consumption.
16. **Game AI:** A* is also used to makeintelligent decisions for non-player characters (NPCs), such as determining the best way to reach an objective or coordinate movements in a team-based game.

AO* Algorithm

The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems. The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.

The start state and the target state are already known in the knowledge-based search strategy known as the **AO* algorithm**, and the best path is identified by heuristics. The informed search technique considerably reduces the algorithm's **time complexity**. The AO* algorithm is far more effective in searching AND-OR trees **than** the A* algorithm.

Working of AO* algorithm:

The evaluation function in AO* looks like this:

f(n) = g(n) + h(n)

f(n) = Actual cost + Estimated cost

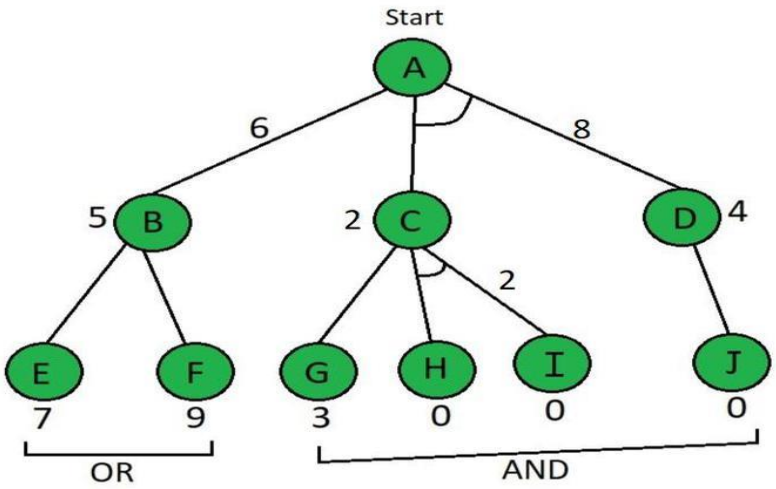
here,

f(n) = The actual cost of traversal.

g(n) = the cost from the initial node to the current node.

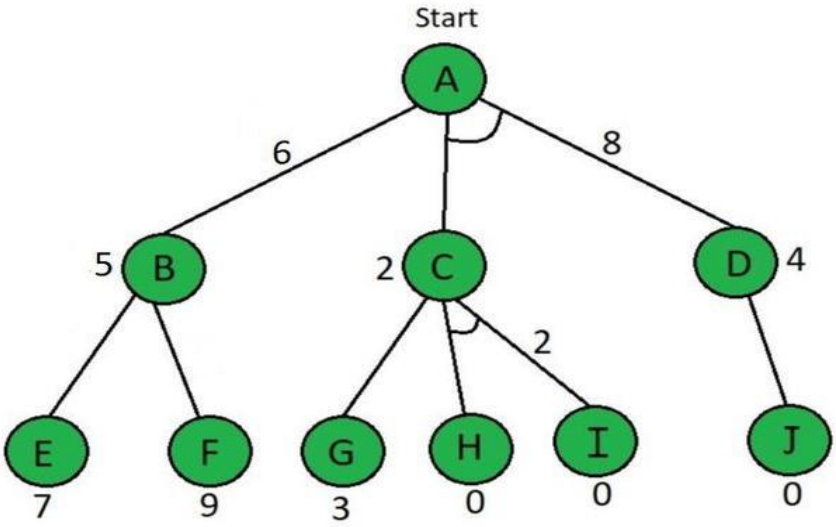
h(n) = estimated cost from the current node to the goal state.

Example



Here in the above example below the Node which is given is the heuristic value i.e h(n). Edge length is considered as 1.

Step 1



With help of f(n) = g(n) + h(n) evaluation function,

Start from node A,

f(A⇢B) = g(B) + h(B)

= 1 + 5 ……here g(n)=1 is taken by default for path cost

= 6

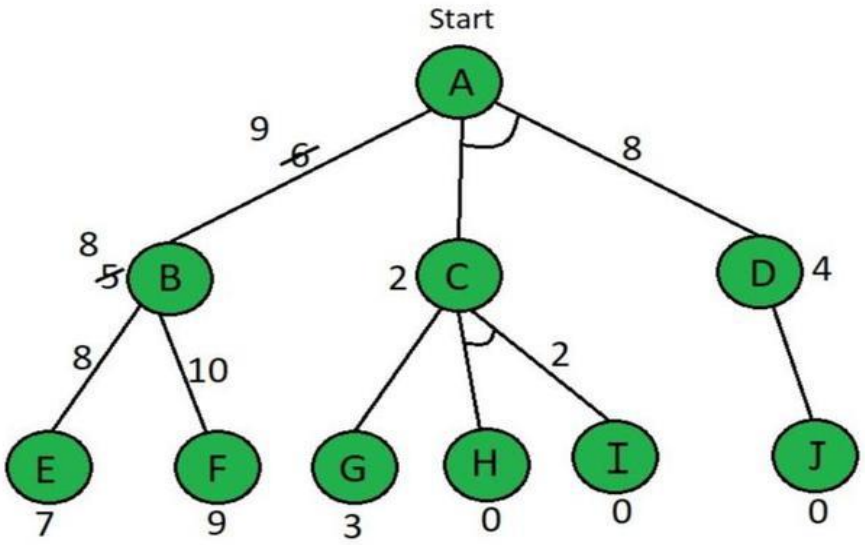
f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)

= 1 + 2 + 1 + 4 ……here we have added C & D because they are in AND

= 8

So, by calculation A⇢B path is chosen which is the minimum path, i.e f(A⇢B)

Step 2



According to the answer of step 1, explore node B

Here the value of E & F are calculated as follows,

f(B⇢E) = g(e) + h(e)

f(B⇢E) = 1 + 7

= 8

f(B⇢f) = g(f) + h(f)

f(B⇢f) = 1 + 9

= 10

So, by above calculation B⇢E path is chosen which is minimum path, i.e f(B⇢E)

because B's heuristic value is different from its actual value The heuristic is

updated and the minimum cost path is selected. The minimum value in our situation is 8.

Therefore, the heuristic for A must be updated due to the change in B's heuristic.

So we need to calculate it again.

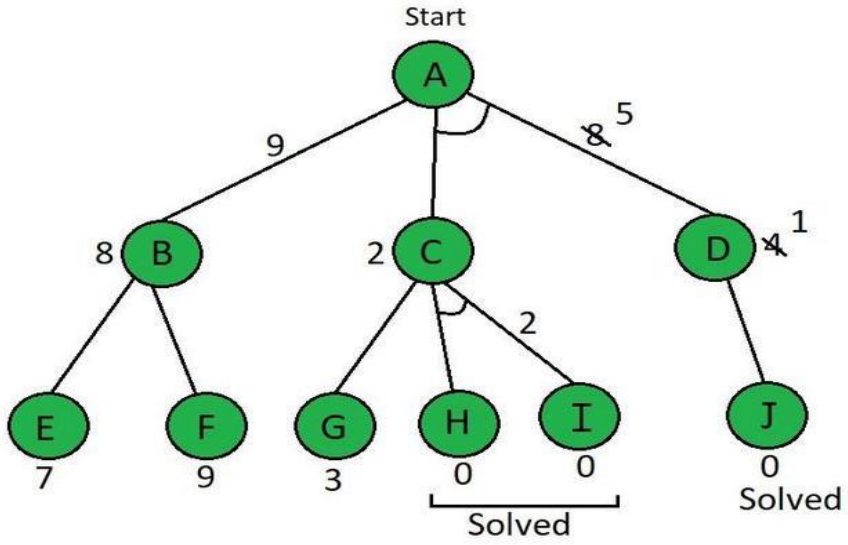
f(A⇢B) = g(B) + updated h(B)

= 1 + 8

= 9

We have Updated all values in the above tree.

Step 3



By comparing f(A⇢B) & f(A⇢C+D)

f(A⇢C+D) is shown to be smaller. i.e 8 < 9

Now explore f(A⇢C+D)

So, the current node is C

f(C⇢G) = g(g) + h(g)

f(C⇢G) = 1 + 3

= 4

f(C⇢H+I) = g(h) + h(h) + g(i) + h(i)

f(C⇢H+I) = 1 + 0 + 1 + 0 ……here we have added H & I because they are in AND

= 2

f(C⇢H+I) is selected as the path with the lowest cost and the heuristic is also left unchanged

because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is 0,

but Path A⇢D needs to be calculated because it has an AND.

f(D⇢J) = g(j) + h(j)

f(D⇢J) = 1 + 0

= 1

the heuristic of node D needs to be updated to 1.

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)

= 1 + 2 + 1 + 1

= 5

as we can see that path f(A⇢C+D) is get solved and this tree has become a solved tree now.

In simple words, the main flow of this algorithm is that we have to find firstly level 1st heuristic

value and then level 2nd and after that update the values with going upward means towards the root node.

In the above tree diagram, we have updated all the values.

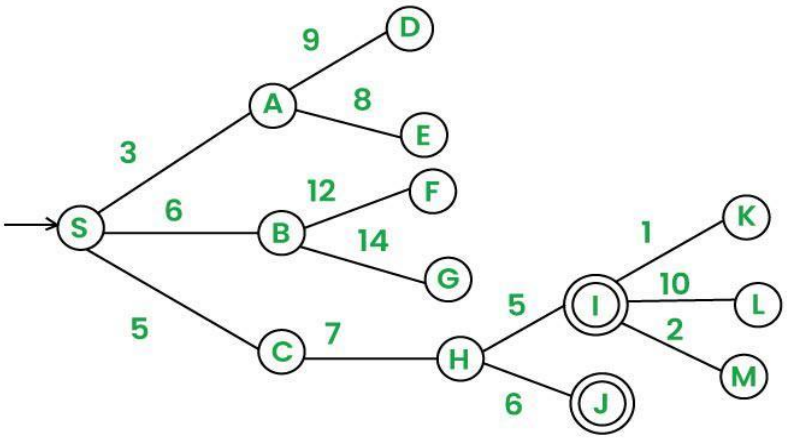
Best First Search

In BFS and DFS, when we are at a node, we can consider any of the adjacent as the next node. So both BFS and DFS blindly explore paths without considering any cost function.

The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.

Best First Search falls under the category of Heuristic Search or Informed Search.

Example



Best First Search (Informed Search)

- We start from source “S” and search for goal “I” using given costs and Best First search.

- Priority Queue(pq) initially contains S
    - We remove S from pq and process unvisited neighbors of S to pq.
    - pq now contains {A, C, B} (C is put before B because C has lesser cost)

- We remove A from pq and process unvisited neighbors of A to pq.
    - pq now contains {C, B, E, D}

- We remove C from pq and process unvisited neighbors of C to pq.
    - pq now contains {B, H, E, D}

- We remove B from pq and process unvisited neighbors of B to pq.
    - pq now contains {H, E, D, F, G}
- We remove H from pq.
- Since our goal “I” is a neighbor of H, we return.

Game Search

Game Playing is an important domain of artificial intelligence. Games don’t require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve –

- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

Game playing is a popular application of artificial intelligence that involves the development of computer programs to play games, such as chess, checkers, or Go. The goal of game playing in artificial intelligence is to develop algorithms that can learn how to play games and make decisions that will lead to winning outcomes.

1. One of the earliest examples of successful game playing AI is the chess program Deep Blue, developed by IBM, which defeated the world champion Garry Kasparov in 1997. Since then, AI has been applied to a wide range of games, including two-player games, multiplayer games, and video games.

There are two main approaches to game playing in AI, rule-based systems and machine learning-based systems.

1. Rule-based systems use a set of fixed rules to play the game.
2. Machine learning-based systems use algorithms to learn from experience and make decisions based on that experience.

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Formalization of the problem:

A game can be defined as a type of search in AI which can be formalized of the following elements:

- Initial state: It specifies how the game is set up at the start.
- Player(s): It specifies which player has moved in the state space.
- Action(s): It returns the set of legal moves in state space.
- Result(s, a): It is the transition model, which specifies the result of moves in the state space.
- Terminal-Test(s): Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- Utility(s, p): A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

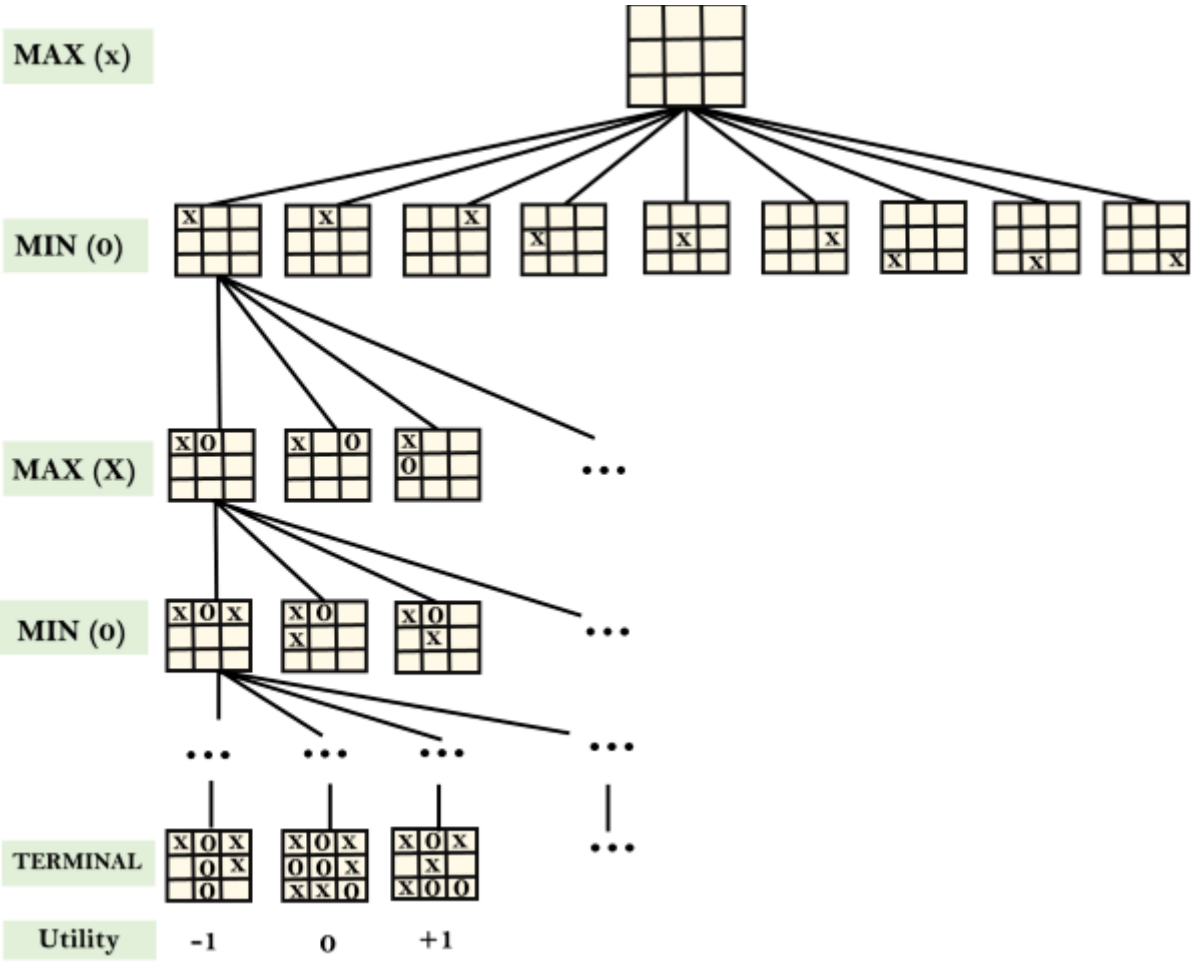
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.

- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$